A Reinforcement Learning Framework for Autonomous Eco-Driving

A Thesis

By

Jianzong Pi

Department of Electrical and Computer Engineering

The Ohio State University

2020

Thesis Defense Committee:

Dr. Abhishek Gupta, Advisor

Dr. Parinaz Naghizadeh

Abstract

Eco-driving involves adaptively changing the speed of the vehicle to ensure minimal fuel consumption. We pose this problem within the framework of Markov decision problem with discounted reward. The key difficulty lies in identifying the state space and the reward function of the vehicle to be able to use reinforcement learning methods so that the vehicle can learn not only the optimal driving strategy, but also the rules of the road through reinforcement learning method. We use deep Q learning and enhanced policy iteration to determine the optimal driving strategy.

Dedicated to my family and friends...

Acknowledgments

First, I want to thank my advisor, Dr.Abhishek Gupta for helping me understanding the theoretical problems and giving me advises while developing the simulator and experiments. I could not have achieved these without help from Dr.Gupta.

I also want to thank my parents for funding my education. I want to thank my friends for always supporting me. I would like to thank Jayanth, Gaurav and Hao for being good teammates while developing CATS. I am indebted to them for their contribution to the software. The process of learning the algorithms and developing the software was one of the most valuable experience in my undergraduate years.

Vita

May 27, 1997 ..... Born - Tianjin, China

Aug 2017 - present ..... B.S. Electrical and Computer Engineering

Fields of Study

Major Field: Electrical and Computer Engineering

Table of Contents

Page

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

In this thesis, we focus on obtaining the optimal driving policy in a straight road with other cars, stop signs and traffic signals. Instead of doing an end-to-end training, we assume the information of the environment, as well as the vehicle, are precisely acquired with multiple sensors. With various machine learning approaches, we can obtain the optimal driving policy in a well-defined environment, state space and reward system.

1.1 Background

Autonomous driving is a prevalent and demanding application in the field of machine learning. 70% of traffic accidents are caused by human errors [7]. Autonomous vehicles are expected to reduce the accident rate dramatically. However, not all accidents could be avoided. Autonomous vehicles sometimes are still required to make ethical decisions for the unavoidable harm[3]. The design of the decision system must follow codes of engineering ethics to maximize social utility.

Environment recognition, action planning and prediction are important for an autonomous vehicle. [8][5] proposed the autonomous driving problem as a discrete state, discrete action Markov Decision Problem (MDP). In this project, we may assume that

the raw data from multiple sensors are processed, and we focus on approaches to the optimal policy to achieve safe driving, time efficiency and fuel saving.

Chapter 2: Algorithms

In this section, we discuss the algorithms we used for training. We start by discussing Markov Decision Process. Then we discuss Q learning algorithm. Finally, we discuss about enhanced policy iteration.

2.1 Markov Decision Process

Markov decision process (MDP) is a discrete time stochastic process. Consider an MDP with finite state and action space, transition probability P_{ij}^a and reward R_{ij}^a are used to specify the dynamics of the system[6]. $r_{t+1} \in \mathbb{R}$ is the reward the agent gets at timestep $t+1$. The transition probability and the reward are denoted as

$$P_{ij}^a = P\{s_{t+1} = j | s_t = i, a_t = a\}$$

and

$$R_{ij}^a = \mathbb{E}\{r_{t+1} | s_t = i, a_t = a, s_{t+1} = j\}$$

Let $\mathbf{S}$ and $\mathbf{A}$ denote the state and action space respectively. Value function $V : \mathbf{S} \to \mathbb{R}$ shows the expected reward the agent could get given policy π. The value function is given below[6]

$$V^\pi(s) = \mathbb{E}_\pi\{R_t | s_t = s\} = \mathbb{E}_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\}$$

Where R_t is the cumulative reward starting at time t, and γ is the discount factor. The optimal value of initial state S is:

$$V^*(s) = \max_{\pi \in \Pi} V^\pi(s)$$

Policy π is optimal if $V^\pi(s_0) = V^*(s_0)$. Similarly, the action value function under policy π, $Q^\pi : \mathbf{S} \times \mathbf{A} \to \mathbb{R}$ is defined as

$$Q^\pi(s, a) = \mathbb{E}_\pi\{R_t | s_t = s, a_t = a\}$$

Denote t_{max} as the termination time step in one episode. The state transition equation at time step t is as follows:

$$s_{t+1} = f(s_t, a_t), t \in \{0, 1, 2, ..., t_{max}\}$$

Q-Learning is an off-policy, model-free reinforcement learning algorithm. The learning process is based on the method of temporal difference(TD)[6] and the algorithm is proved to converge to the optimal factors Q^* with probability 1 [1]. The process of one episode is an undiscounted finite horizon problem with bounded time steps. The Markov decision problem is formulated to derive the optimal policy. The goal of the agent is to achieve the highest cumulative reward, $\sum_{t=0}^{t_{max}} R(s_t, a_t)$.

2.2 Q Learning and Deep Q Network (DQN)

In problems with large state and action space, we use a neural network as a function approximator for the Q functions due to limited memory space for the large-scale reinforcement learning problem. Deep-Q-Network(DQN) combines Q learning with deep neural networks to approximate Q functions. DQN is a nonlinear function approximator for $Q(s, a)$. During the training, we store the tuples (s_t, a_t, r_t, s_{t+1})

during each time step, and store them into a memory buffer. At a given time interval, we sample a batch of the tuples randomly from memory buffer and use the batch to update parameters of the neural network. The training algorithm for DQN with epsilon greedy as exploration method is shown below.

Algorithm 1 DQN Training Process (epsilon-greedy) [4]

Initialize Q-function Q, target Q-function $\hat{Q} = Q$
while not converging **do**
  Choose random action with probability ϵ, or choose $a_t = argmax_a Q(s_t, a)$
  Take action a and obtain reward r and next state s_{t+1}
  Store tuple (s_t, a_t, r_t, s_{t+1}) in replay buffer
  Sample a mini-batch of tuples from the replay buffer
  Calculate loss $L = (Q_{s,a} - y)^2$, where $y = r$ for terminal states and $y = r + \gamma \max'_a(\hat{Q}_{s',a'})$ otherwise
  Use stochastic gradient descent to train $Q(s, a)$
  For every N steps, copy weights from Q to $\hat{Q}$
end while

Note that different exploration methods such as Boltzmann exploration can be used to choose action to speed up the rate of convergence.

The exploration-exploitation dilemma is a classic problem in reinforcement learning. "Exploration" means getting information by randomly choosing different actions, whereas "exploitation" means choosing the action that gives the highest expected reward. There are several exploration methods that cope with the dilemma. Epsilon greedy is a solution to the exploration-exploitation dilemma. Epsilon greedy makes the agent randomly choose actions (explore) with higher probability when the agent has limited information, and less likely to randomly act when more information is obtained.

It is required in stochastic gradient descent (SGD) that the training data are i.i.d., but in DQN training, the samples (s_t, a_t, r_t, s_{t+1}) are not independent. Moreover, the distribution of training data may not be the same as the ideal samples for the agent to get the optimal policy[4]. To deal with the two problems, a replay buffer with a large size is needed. The sampled data follows a "first-in-first-out" mechanism to maintain the size of memory buffer. During training, the training sample are randomly picked in order to minimize the correlation of the sampled data for SGD.

2.3 Policy Iteration and Enhanced Policy Iteration

2.3.1 Policy Iteration

A policy has to be measurable in order to be improved. In policy iteration, a policy is measured in policy evaluation phase. In this phase, policy is evaluated with

$$V_\pi(s) = \mathbb{E}_\pi[R_t + \gamma V_\pi(s_{t+1})|s_t = s]$$

We usually use linear programming to solve a set of linear equations in this process. The number of equations is equal to the number of states. The policy improvement comes after the evaluation of policy. The "improved" policy is calculated based on the estimated value of states from evaluation phase. The new policy chooses actions with highest expected value of the reward and the old value function. The updated policy is guaranteed to have a better performance than the old one[1].

2.3.2 Enhanced Policy Iteration

Enhanced policy iteration[2] is a policy iteration-like Q learning algorithm. As compared to equation (4), Q^* is computed by a different mapping $F_{V,\mu}$,

$$F_{V,\mu}Q(s,a) = \sum_{s'} \left(p_{ss'}(u)(r(s,a,s') + \gamma \sum_{a'} \mu(a'|s') \max\left(V(s'), Q(s',a')\right)\right) \quad (2.1)$$

The new mapping uses a randomized Q factor $\mu(a'|j)$. Ordinary policy iteration algorithms usually do a linear programming in policy evaluation, whereas enhanced policy iteration solves an optimal stopping problem in this phase. Instead of only iterating over Q in policy iteration, enhanced policy iteration iterates over two factors - Q and V. It was also mentioned that the mapping $F_{V,\mu}Q$ is a sup-norm contraction in [2], which is an important property which policy iteration does not have. Because $F_{V,\mu}$ is a contraction map, fixed point (Q^*, V^*) will be achieved when iteration goes. Similar to Q learning, the algorithm for enhanced policy iteration is

Algorithm 2 Enhanced Policy Iteration Training Process (epsilon-greedy) [4]

Initialize Q-function Q, target Q-function $\hat{Q} = Q$
while not converging **do**
  Choose random action with probability ϵ, or choose $a = argmax_a Q(s, a)$
  Take action a and obtain reward r and next state s'
  Store tuple (s, a, r, s') in replay buffer
  Sample a mini-batch of tuples from the replay buffer
  Calculate loss $L = (Q_{s,a} - y)^2$, where $y = r$ for terminal states and $y = r + \gamma \sum_{a'} \mu(a'|s') \max(V(s'), Q(s', a'))$ otherwise
  Use stochastic gradient descent to train $Q(s, a)$
  For every N steps, copy weights from Q to $\hat{Q}$
end while

Chapter 3: CATS

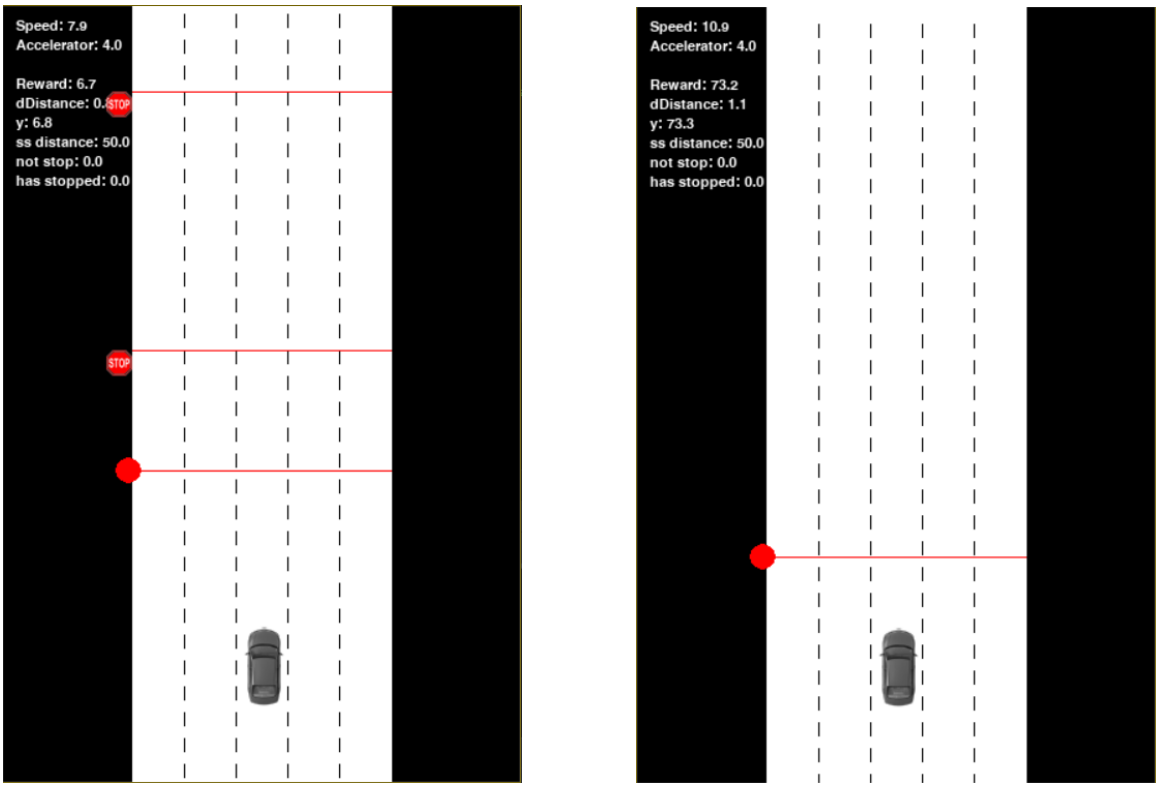
3.1 Introduction

In order to monitor and display the training process easily, we created a simulation software, CATS. CATS is developed in python 3 and can be run on Windows or Linux.

This simulator consists of four parts

- A simulator which simulates a multi-lane infinite straight road. The road has some rule based probabilistic cars and one smart car. The rule based cars follow all traffic rules, but with different levels of aggressiveness (distance to maintain with front car, overspeeding etc). The smart car takes its surrounding as input state and decides on gas/brake.
- An openAI gym style interface to the simulator environment and the smart car
- A template Deep Q-Learning code that uses above 2 to control the acceleration of the smart car.
- PyGame code to display the simulation.

We propose an infinite road model with multiple lanes. The environment contains the agent, multiple rule-based cars and stop signs. The interface is shown in Figure 3.1. The red cars are rule-based cars, and grey car is our agent.



(a) (b)

Figure 3.1: (a), (b) Example environments

3.2 Simulation Setup

The following steps are needed to setup the environment needed to run the project.

- Setup Python 3 virtual environment

```
pip install virtualenv
```

- Create environment

```
virtualenv -p $(which python3) env$
```

- Activate environment

```
source env/bin/activate
```

- Install requirements

```
pip install -r requirements.txt
```

- Run script

```
python dqn_main.py <Name of this simulation>
```

- Display episode

```
python dqn_display.py <Simulation_name ><episode_number>
```

3.3 Directory Structure

The directory structure is as followed -

```
simulator/ (Contains simulator related code)
    media/
        images/
            (Images used in simulation)
```

```
    stuff_on_road/
        normal_car.py (Contains code for rule based cars)
        road.py (Contains road related code)
    utils/
        ds.py (Contains utility functions and custom data structures)
        constants.py (Contains all simulation related constants)
display/ (Contains simulation visualisation related code)
    display.py (Display while running simulation)
    offline_display.py (Display from simulation logs)
rl/ (Contains model architecture)
    agent.py (Agent and the Q-Network classes)
    environment.py (OpenAI gym style environment for the simulator)
    simulator_car_interface.py (Interface to simulator car)
dqn_main.py (Run DQN training)
dqn_main.py (Visualise a particular episode of training)
main.py (Run simulation where smart car just goes straight at speed limit)
```

3.4 Dynamic model of infinite road

We consider an infinite straight road with five lanes. There also exists stop signs at certain position. The road has multiple rule-based cars and one smart car. The rule based cars follow hand-coded traffic rules. Each rule based car has its own aggressiveness. The aggressiveness of the rule based cars decide the distance it maintain with front car and the probability of overspeeding. They also exhibit complex behaviours like overtaking from left only, staying on right unless to overtake, etc.

3.5 State Representation

To achieve a optimal driving policy, the agent should take all the information needed as its observation. To achieve our goal, the state space is a vector

$$S_t = [v_t, a_t, d_t, v_t', f_t]^T$$

where

- v_t is current speed of agent.
- a_t is the current acceleration of the agent.
- d_t is the distance to the front car if there exists a front car in the same lane with the agent, else d_t is set to a the safety distance D_s.
- v_t' is the velocity of the front car if there exists a front car in the same lane with the agent, else v_t' is set as the safe velocity of the car.
- f_t is the fuel consumption over the one time step.

3.6 Admissible Actions of the Vehicle

Denote $A_t = [g_t]$ as the action of the agent at time t. For simplicity for training, we formulate the autonomous driving problem with discrete action space. We assume the max acceleration of the car, $a_{max} = 16$, and the action space as

$$[-a_{max}, -\frac{1}{2}a_{max}, 0, \frac{1}{2}a_{max}, a_{max}]$$

When $g_t < 0$, the car is braking, and when $g_t > 0$, the car is accelerating. A large action space may cause many actions to have negative number before exploration is

biased, which makes the learning process lengthy, and the temporal difference(TD) high.

$$v_t \leq \begin{cases} v_t^f & \text{if } d_t^f \leq D_s \text{ and } v_t^f \leq \bar{v}_t \\ \bar{v}_t & \text{otherwise} \end{cases}$$

3.7 State Update Equation of the Vehicle

The speed of vehicle follows the following state update rule:

$$v_{t+\delta t} = v_t + C_p * g_t * \delta t - C_d * v_t^2 * \delta t$$

C_p and C_d represent the acceleration and drag coefficient. The distance to front car follows:

$$d_{t+\delta t}^f = \begin{cases} d_t^f + (v_t^f - v_t) * \delta t & \text{if there exists a front car} \\ D_s & \text{otherwise} \end{cases}$$

3.8 Reward Function of the Vehicle

A good reward function is necessary to make the convergence process faster. The fuel consumed at time t is determined by a function $f : \mathbb{A} \to [f_{min}, f_{max}]$, where $0 < f_{min} < f_{max} < \infty$, and $\mathbb{A} = \{-a_{max}, -\frac{1}{2}a_{max}, 0, \frac{1}{2}a_{max}, a_{max}\}$. The reward function, $R : S \times A \to \mathbb{R}$ is defined as:

$$r_t = v_t - f_t + \alpha * sc + \beta * sv + \delta * ss$$

Where

- sc is true when the learning agent is collided with other cars or objects
- sv is true when the learning agent has violated traffic rules (overspeed)
- ss is true when the speed of the learning agent is less than 0.1

Chapter 4: Simulation

The simulation includes three parts - simulator, controller and decision-maker. The simulator provides an environment for training. The environment provides "observation" - all information in the environment to the controller. The controller filters observations and gives the decision maker only the information needed for training, the controller takes the "actions" from decision-maker and executes the "actions" in the environment.

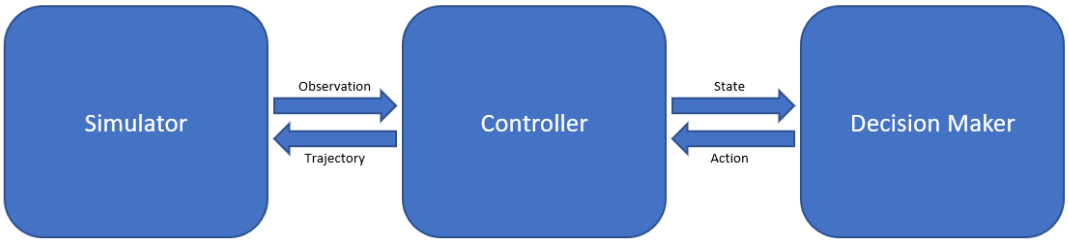


Figure 4.1: Simulation Framework

The simulator was developed under Python 3 and tensorflow 1.15.x. The simulation was run on Chameleon cloud with Skylake microarchitecture CPUs with 64 kB L1 cache and 256 kB L2 cache and 4.5GHz maximum clock rate. 1000 Episodes of training take approximately 15 hours to finish.

The structure of the DQN is a 2 layered fully connected neural network with 50 neurons at each layer. ReLU is used for hidden layers, and we use softmax as the output layer. We use Adam optimizer with 0.005 learning rate and a replay buffer of size 2500. The target network is asynchronously replaced with the evaluation network every 15000 time steps. We set $\beta = 0.1$ and $k_{max} = 5000$ for each episode. During the learning process, we randomly sample from replay buffer to decrease serial correlations between training tuples (s, a, r, s'). We set the other parameters as follows

Parameter	Value
α	-100
β	-20
δ	-0.5
$\bar{v}$	25
D_s (safety distance)	50
C_p (acceleration coefficient)	1
C_d (drag coefficient)	0.005
f_{max} (max fuel consumption per timestep)	0.1
f_{min} (min fuel consumption per timestep)	0.02

Table 4.1: Parameter values

	DQN	Enhanced Policy Iteration
Average score	80.89	72.39

Table 4.2: Performance

It turns out hard for the agent to learn to behave properly at stop signs besides efficient driving. To deal with this issue, rules are set while agent approaches the stop

signs, thus the length of state vector can be as short as 5. As a result, the effect of training could be observed within 1000 episodes.

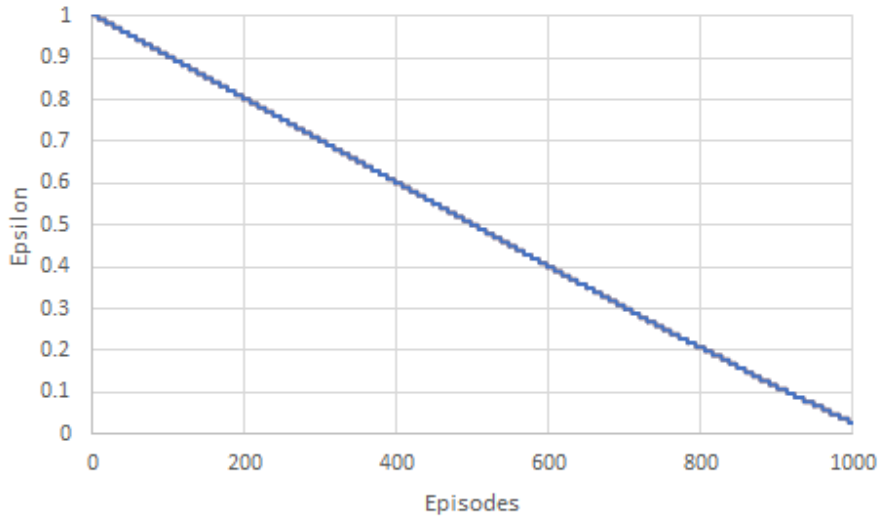


Figure 4.2: Epsilon vs. Episodes

In both experiments, epsilon-greedy was chosen as the exploration method, and ϵ was linearly decreasing with episodes. Starting from 1, ϵ decreases to 0.01 at the end of the training. Similar to DQN, we also use a function approximator with same structure for $Q(s, a)$. However, we change the mapping of Q functions during training, and follow Algorithm 2 in section 2.3.2 to perform the training. For simplicity, $\mu(a'|s')$ is chosen to be uniform distribution. An upward trend of average reward could be seen while using both algorithms, and both results show a large variance of cumulative reward, especially during the last 200 episodes. One possible reason may be inadequate exploration was conducted before ϵ goes low. Decreasing the learning rate, and increasing the training episodes will improve this situation.

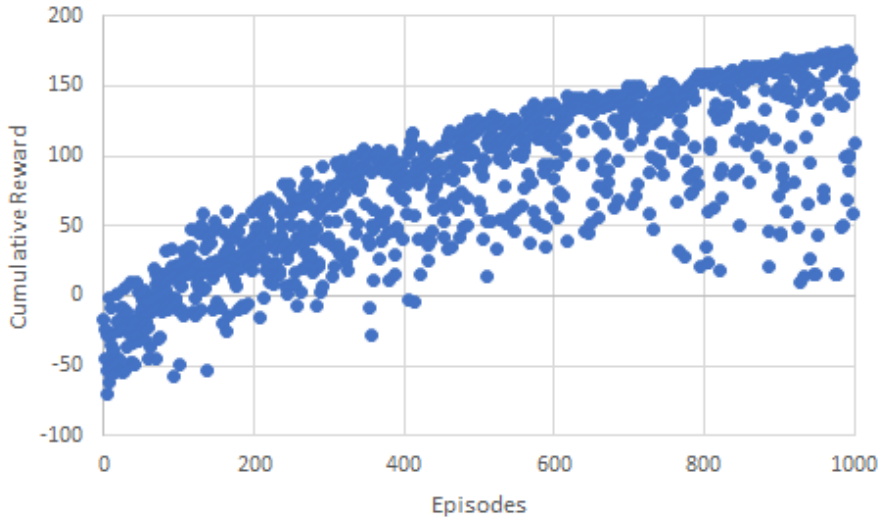


Figure 4.3: DQN - Average reward vs. Episodes

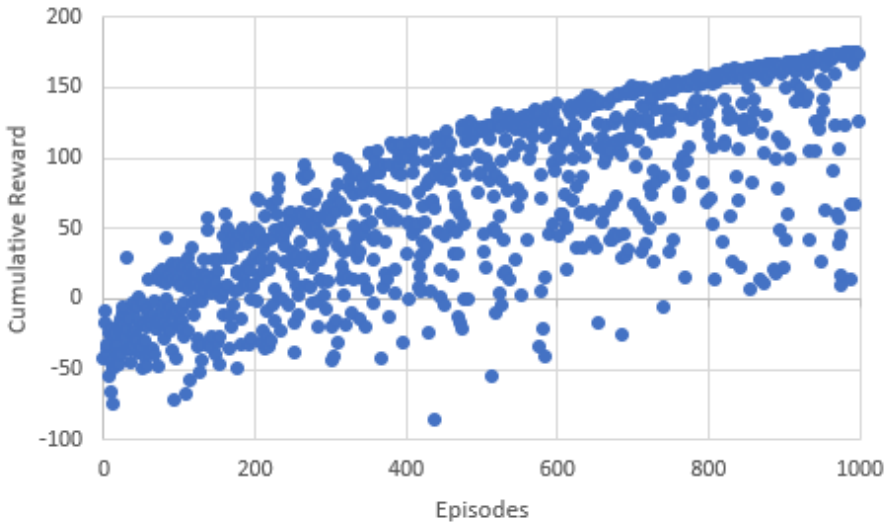


Figure 4.4: Enhanced Policy iteration - Epsilon vs. Episodes

Chapter 5: Conclusion and Future Work

5.1 Conclusion

A reinforcement learning framework for autonomous driving was established. Proper state space and reward system were chosen and tuned. Deep Q learning and enhanced policy iteration were used for learning. A proper policy could be gained after 1 million (s, a, r, s') tuples were sampled and used for training, as we could see the upward trend of average reward gained over training in figures 4.3 and 4.4.

5.2 Future Work

First, exploration methods other than epsilon-greedy could be used to balance exploration and exploitation. In epsilon-greedy, only the action with the highest expected reward was chosen with a relatively high probability while other actions with a small probability. Other than epsilon-greedy, upper confidence bound(UCB) exploration method could be utilized to take both the current estimates as well as the uncertainties in the estimates.

Second, a probability distribution μ other than uniform distribution need to be picked in equation 2.1. Picking a good distribution may yield a better solution of the optimal stopping problem, thus accelerate the training process.

Third, policy gradient methods could solve RL problems with large state and action space. Based on the Openai leaderboard, policy gradient methods outperform value iteration methods, especially in continuous state and action problems. Training the agent for more complicated tasks can be possible with policy gradient methods.

Last, since we are using a squared loss as our loss function to train the neural network, the contraction for the function approximator is a L^2 type contraction. However, the contraction F we used while implementing enhanced policy iteration is an inf-norm contraction. As a result, there is no theoretical guarantee of contraction that leads to convergence. We can use a loss kernel that satisfies inf-norm contraction.

Bibliography

[1] Dimitri P Bertsekas and John N Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

[2] Dimitri P Bertsekas and Huizhen Yu. Q-learning and enhanced policy iteration in discounted dynamic programming. *Mathematics of Operations Research*, 37(1):66–94, 2012.

[3] Jean-François Bonnefon, Azim Shariff, and Iyad Rahwan. The social dilemma of autonomous vehicles. *Science*, 352(6293):1573–1576, 2016.

[4] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[5] Subramanya Nageshrao, H Eric Tseng, and Dimitar Filev. Autonomous highway driving using deep reinforcement learning. In *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pages 2326–2331. IEEE, 2019.

[6] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press Cambridge, 1998.

[7] John R Treat. A study of precrash factors involved in traffic accidents. *HSRI Research Review*, 1980.

[8] Junjie Wang, Qichao Zhang, Dongbin Zhao, and Yaran Chen. Lane change decision-making through deep reinforcement learning with rule-based constraints. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6. IEEE, 2019.